記　号　処　理 77－2
（１９９５．１．１３）

パイプライン型アーキテクチャにおける OR並列型Prolog実行の一検討

稲葉 勉　沈 紅　片平昌幸　小林広明　中村維男

東北大学情報科学研究科

本研究は、Prolog処理の高速化を目的とした、パイプライン型アーキテクチャ上でのOR並列Prologのコードブロックレベルの並列処理手法を提案する。これは、J.Beerの提案による逐次型Prologのパイプライン実行をOR並列Prolog用に拡張したものである。拡張に伴い、大域共有メモリ、クロスバスイッチを導入した。大域共有メモリは、選択点履歴を格納する選択点スタックモジュールと、各選択点での環境フレームが均等に配置されるモジュールから構成される。本論文では、システムの概要を述べ、ベンチマークプログラムによる性能評価を行う。性能評価の結果、逐次型Prologのパイプライン実行と比較して約2.5倍のLIPS値を得ることができた。

Pipelined Execution of OR-Parallel Prolog

Tsutomu Inaba, Hong Shen, Masayuki Katahira, Hiroaki Kobayashi, Tadao Nakamura

Department of Computer and Mathematical Sciences
Graduate School of Information Sciences
Tohoku University
Aramaki Aza Aoba, Aoba-ku, Sendai 980-77, Japan

In this study, we propose an OR-Prolog parallel execution model on a PE-pipeline architecture. This is an extention of J.Beer's idea in "Pipelined Execution of Sequential Prolog." On our model, we adopt global shared memory and crossbar network. The Global shared memory that consists of a Choice-Point Stack Module and several Environment Frame Modules that store the environments of each Choice-Point. In this paper, the system organization and simulation results are described. Based on the simulation results, we can obtain the LIPS 2.5 times of that on Beer's model.

1. はじめに

論理型言語Prologは、述語論理をベースにした記号処理向き言語であり、推論機能を持つ、プログラムを簡潔に記述できるなどの理由から知識情報処理分野での幅広い応用が期待されている。しかしながら、Prologは人工知能向き言語として優れた特性を持つ反面、汎用コンピュータで実行した場合、実行速度が遅いこと、大容量のメモリを必要とすることが問題とされてきた。

これらの問題は、Warrenの偉大な功績であるWAM[1]等により、逐次処理方式においては一応の解決をみた。しかしながら、莫大な計算を要求する問題を解決するために、現在は並列処理による高速化が注目されている。並列処理では、実行前に探索木を予測することが困難であるため、負荷分散並びに拡大する探索空間を表現するための実行環境管理が非常に難しい。

これらの研究の一つとして、逐次Prologのパイプライン処理[2]が提案されている。このシステムは、コードブロックレベルでのパイプライン処理を行い、スタック操作、選択点生成、データ受け渡し等の中核となるオーバーヘッドを削減している。我々は、[2]のPEをパイプライン状に結合したアーキテクチャに、OR並列Prologをマッピングすることでさらなる高速化が得られると考えた。これによると、OR並列とコードブロックパイプライン処理の並列性が最大限に利用できる。

本稿では、システムのOR並列化に伴う改良点、システムの負荷を均一に分散するためのタスクマッピング法、システムオーバーヘッドの中核をなすデータ管理法について述べ、最後にベンチマークテストによる性能評価を行う。

2. 論理型言語の並列性

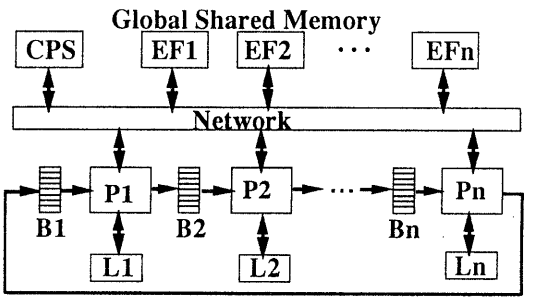
Prologプログラムの実行は、節中のゴールの処理を左から右へ一つずつ行う。この際、呼び出しゴールと単一化可能な節が複数ある場合は上から順に選択される。このように、Prologの計算過程は論理型プログラムの一つの逐次実行モデルであるが、複数のプロセッサを持つマシンでは、逐次的に実行せず、並列に実行しても正しい解を生成することが可能である。Prologの並列処理方式には、AND並列、OR並列、ストリーム並列などがあるが、本研究ではOR並列性に着目する。OR並列実行では、同時に存在し得る複数のOR分岐の変数束縛環境の実現が難しいことが指摘されている[3]。

3. OR並列Prologのパイプライン処理

本稿で提案するOR並列Prologのパイプライン処理は、逐次Prologのパイプライン処理[2]のProlog逐次実行をOR並列展開し、その処理過程をパイプライン型アーキテクチャにマッピングしたものである。OR並列拡張をサポートするために、[2]で提案されているアーキテクチャに対して、共有メモリ、ネットワークに変更を加えた。

3.1 システムアーキテクチャ

図1にシステムアーキテクチャの構成を示す。この実行モデルは、Prologの逐次実行を対象として[2]で提案されているPEをパイプライン状に結合した共有メモリ型実行モデルを拡張したものである。プログラムをコンパイルして得た命令コードはローカルメモリ内にコピーされ、マイクロプログラム制御で実



CPS: Choice point Stack Module
EFi : Environment Frame Module
Pi : Processing Element
Bi : Pipeline Buffer
Li : Local Memory

図1 システム構成

行される。PEを連結するパイプラインバッファは、固定サイズのメモリブロックからなるリングバッファである。パイプラインバッファの各ブロックは、制御ワードと引数レジスタで構成される。PE間のデータ転送は、左側から右側への単方向で行われ、共有メモリへの参照はポインタの受け渡しにより行われる。従って、PE間の通信量は軽減される。また命令コードを指すポインタ転送によりPE間の同期も制御される。大域共有メモリは、一つの選択点の履歴をとる選択点スタック用のモジュールと、各選択点での環境を記録する複数のモジュールで構成される。

3.2 コードブロックのパイプライン処理

パイプライン処理は命令を処理単位に分割し、それらをオーバーラップさせ処理の高速化を得る方法である。パイプライン処理は、処理単位の大きさにより以下に示すような、さまざまなレベルで実現されている。

(1) マイクロ命令のパイプライン処理
(2) マシン命令のパイプライン処理
(3) コードブロックのパイプライン処理

我々は、複数の独立な単一化処理を並列実行するために、パイプライン型アーキテクチャ上で、コードブロックレベルでのパイプライン処理を行う。

例として、3つのPEでのパイプライン処理実行を図2に示す。PEでの処理単位は節呼出しとリターン処理であり、同図中の数字は実行の順序を示している。命令コード中にあるAn*並びにAnはそのPEから見てそれぞれ右隣、左隣のパイプラインバッファ内の引き数レジスタである。各PEはそれぞれ各自の実行結果をパイプラインバッファに書き込み、次のPEはパイプラインバッファの情報を基に実行を継続する。同図よりPEで生成されたデータが次々に次PEへ送信され、連続して並列に処理が行われていることがわかる。

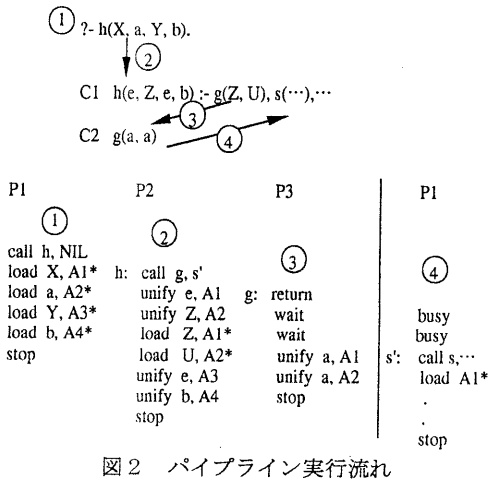


図2 パイプライン実行流れ

3.3 負荷分散法

各プロセッサでの処理単位は

(1) 述語呼び出しと単一化
(2) 前環境へのリターン処理

である。(1)、(2)の処理の大きさはプログラムによって異なるが、比較的大きさの等しい処理単位をもつ逐次Prologのパイプライン処理では図3のような実行が可能である。しかしながら、(1)や(2)の処理単位は小さく、プロセッサ台数を増やしても台数効果が少ないことがわかる。そこで、図3の空白部に処理を割り当て台数効果を上げるため、OR並列

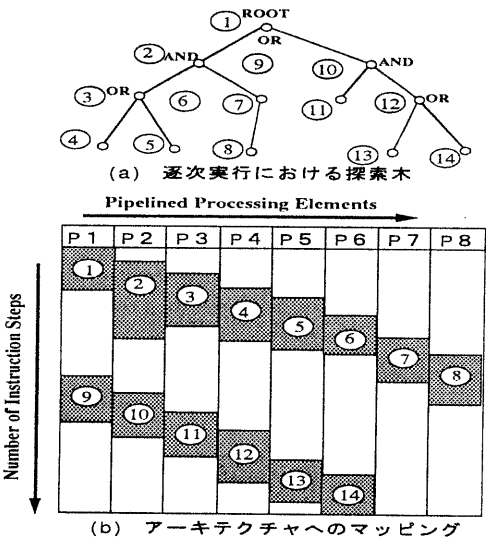


図3 逐次Prolog処理のパイプライン型アーキテクチャへのマッピング

Prologをマッピングしプロセッサのアイドル時間を減少させたものが、本処理方式である。探索木とプロセッサへの処理のマッピング状態を図４に示す。OR分岐を呼び出すプロセッサは、その時点での環境を次のプロセッサに継承し、そこで全部のOR分岐の節呼び出しが幅優先探索で行われる。したがって、すべてのOR分岐が同じパイプラインバッファを見て処理を行うために同じ環境を再生する必要がない。また、単一化に失敗した処理はそこで中止され、バックトラックも発生しないという利点がある。(2)のリターン処理は逐次Prologの場合と同様に行う。

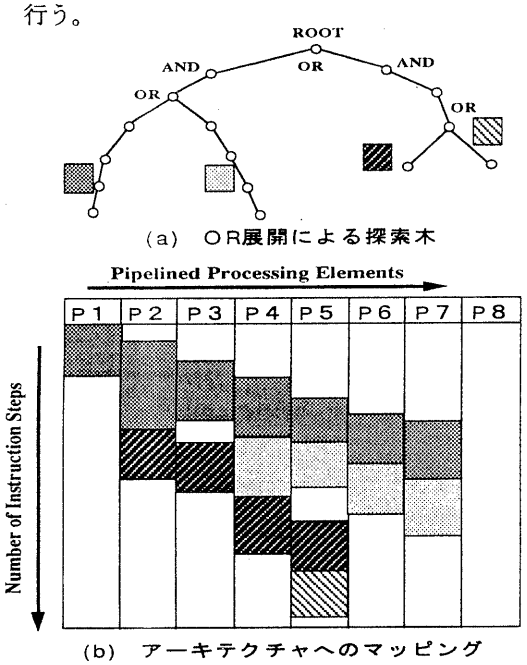


図４　OR並列Prolog処理のパイプライン型アーキテクチャへのマッピング

3.4　コンパイル時の静的負荷分散

OR分岐はすべて次段のPEに継承され、同一のレジスタ情報を共有して実行される。しかし、OR分岐が大量に発生する場合は、処理が一つのプロセッサに集中し全体的な負荷のバランスを大きく崩す恐れがある。こうした事態を回避するため、コンパイル時に静的な負荷分散を行う。

Prologプログラムは、一般に有限なデータ構造から情報を取り出す、あるいは処理を行う他に、再帰データ型を用いて再帰的処理を行う。従って、プログラムに出現する各手続きは、少数の規則節と多数の事実節で構成されることが少なくない（事実節も少数の場合もある）。これらの理由から、事実節のみ数個ずつブロック化し負荷を次のPEに先送りし、PEの負荷均等化を行う。事実節のブロック化は命令コードに簡単な変更を加えることで実行が可能である。

3.5　パイプラインバッファ

パイプラインバッファは、プロセッサ間のデータ交換を、同期・緩和させる役目を持つN個の固定サイズのメモリブロックからなるリングバッファである。各ブロックは、図５に示すように、５つの制御ワードと、次の手続き呼び出しで用いる引数を受け渡すための引数レジスタで構成される。パイプラインバッファは、32ワードのブロック8個（計256ワード）から構成される。各ブロックでは5つの制御ワード領域を有するので、引数領域は27個となる。実用的なプログラムの場合、一つの節に出現する引数は、大きなBodyを持つ節でも高々10個程度であり、27の引数領域で十分であると考えられる。引数が27個を超えた場合は、環境フレームを介してデータを交換する。

リングバッファの各ブロックへのアクセス制御は、パイプラインバッファをはさむプロセッサが、それぞれ所有する2つのポインタを用いて行う。

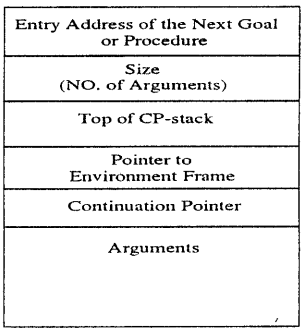


図５　パイプラインバッファ構造

4. 環境管理法

OR並列実行化にともなう多重環境の大域的なデータを、大域共有メモリが管理する。大域共有メモリは、選択点スタックモジュールと、環境フレームを格納するモジュールから構成される。

4.1 選択点スタック

選択点スタックは、OR分岐による選択点の履歴を記録するスタックである。格納される選択点フレームは、OR分岐時のみ生成され、前選択点へのポインタと現時点での環境フレームへのポインタのみを記録する。このモジュールへのアクセス頻度は高いと予想されるが、アクセス時間は短くて済むという特徴を持つ。

4.2 環境フレーム

環境フレームは、各選択点における変数束縛状況を格納するフレームであり、OR分岐発生時に選択点フレームとともに生成され、環境フレームモジュール内に均一に分散される。変数束縛状況の格納にハッシュウインドウ法[3]を応用するため、環境フレームをハッシュ表で表現する。ハッシュウインドウ法は、変数アクセスを一定時間で行うことを目的として、変数アドレスをキーとしたハッシュ表に変数値を登録する方法である。また、変数のデリファレンス発生時には、先頭に位置するフレームから順に参照を行い、見つかったところで自分のフレームに登録する。従って、最悪の場合は先頭フレームまで辿ることになり、従来の手法とかわりない。しかし、ハッシュウインドウ法を採用したPEPSys処理系[4]によれば、大半の場合、1、2回の参照で済むという報告がなされており、本手法でも一定時間での変数アクセスが期待できる。ハッシュ表の役目をする環境フレームは、固定サイズにする必要があるため、メモリ消費量が大きくなるという問題がある反面、再利用しやすいという利点もある。

5. 性能評価

5.1 シミュレータ

OR並列Prologのパイプライン処理の性能を評価するため、図1のアーキテクチャに基づくシミュレータを構築し、ベンチマークテストを行った。本シミュレータの特徴を以下に示す。

(1) [2]と同様なPEを使用し、レジスタトランスファレベルでの評価である。

(2) 命令コードは[2]に基づくが、OR並列化に伴い少量の変更を加えた。

(3) PEの細部まで考慮していないため、各命令コードでのマイクロ命令ステップ数は[2]の値の最悪値とした。

(4) シミュレーションパラメーターは
Machine Cycle= 60 ns
Data Access = 120 ns
Local memory = 60 ns である。

(5) パイプラインバッファは、ブロック数の制限を行わず、キューとして無限に連結される。

本シミュレータは、メモリアクセスの衝突や、実行命令の完全なフェッチ、デコード、実行サイクルを考慮しているため、詳細な結果が期待できる。ただし、本シミュレータは、現在のところコードブロックレベルのパイプライン処理が行われていないため、処理の分散による効果のみが計測される。

5.2 シミュレーション結果、及び考察

ベンチマークプログラムとして、今回はN-QUEENを取り上げた。N-QUEEN問題は記号処理言語の代表的なベンチマークプログラムであり、Nが大きいほど探索のアルゴリズムの並列度は爆発的に増加することが知られている。また、明らかにNが3以下の場合は解を持たない。

図1のアーキテクチャから、本システムはグローバルメモリへのアクセスが、オーバーヘッドになることが予想される。そこでメモリアクセスによるオーバーヘッドを計測する

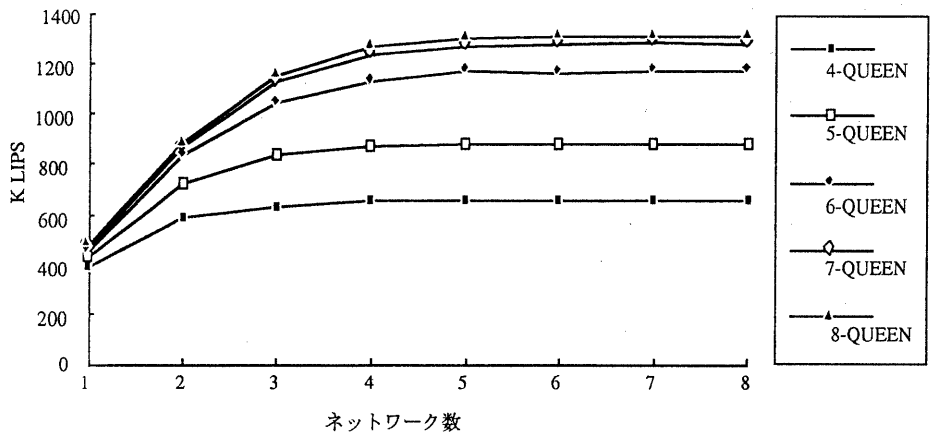


図６　ネットワークと実行速度

ために、プロセッサ8台、環境フレームモジュール8個による条件で、N-QUEEN問題の実行を行った。結果を図６に示す。

横軸のネットワーク数Nは、NxNのクロスバスイッチを仮定している。従って、N=1は単一バスを意味する。縦軸は、１秒あたりの論理推論回数(Logical Inference Per Second = LIPS)である。単一化を試みた処理は、その単一化に失敗した場合でも1推論として計算している。図６によれば、ネットワークの効果が意外に少ないことがわかる。おおよそ6x6のクロスバスイッチを用いれば、ネットワークがボトルネックになることはない。大域メモリのアクセス量の減少には２つの理由が考えられる。１つは、パイプラインバッファによる効果、１つは選択点スタックと環境フレームを異なるメモリモジュールに分割したメモリ構造による効果である。そこで、プロセッサ8台、クロスバ8x8で実行した場合の、各モジュールへのアクセス率を図７に示す。アクセス頻度は高いが1アクセスに要する時間の小さい選択点スタックモジュールと、アクセス頻度は低いが1アクセスに要する時間の大きい環境フレームモジュールでは、選択点スタックモジュールが1.5倍ほど稼働率は高い。しかし、選択点スタックモジュールは、アクセス集中が予想されていたため、得られた値に対するメモリ構造の効果

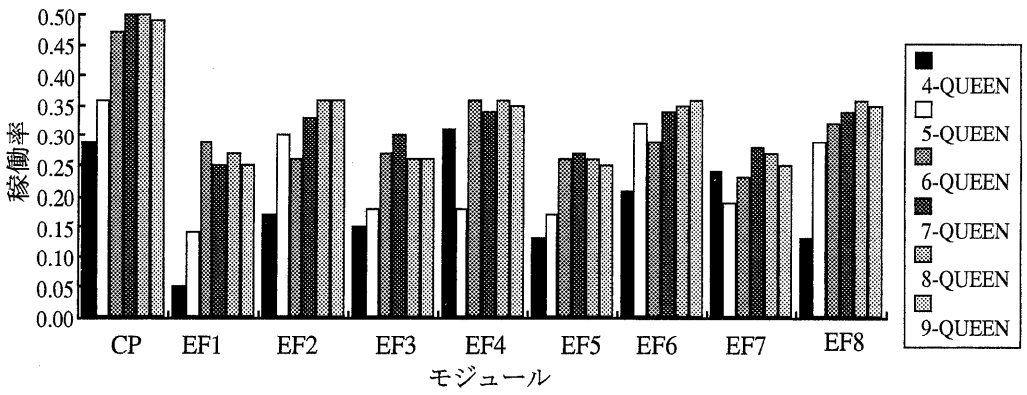


図７　メモリモジュールのアクセス率

は大きいといえる。

次に、プロセッサ台数に関する効果を計測するため、プロセッサ台数を増加させた時のシステムのLIPS値を図８に示す。図８は、プロセッサ台数をNとした時の、黒塗の部分は8×8のクロスバスイッチを、白塗の部分はN×Nのクロスバスイッチを利用した際の8-QUEEN問題での性能である。さらに環境フレームモジュールは、黒塗が8個、白塗はN個である。また、比較のために逐次Prologのパイプライン処理[2]のプロセッサ８台（単一バス使用時）の結果を＃に示す。[2]ではWAM処理系のメモリ構造を持つ逐次処理系であるため、単一バス以外は通信網は意味がない。

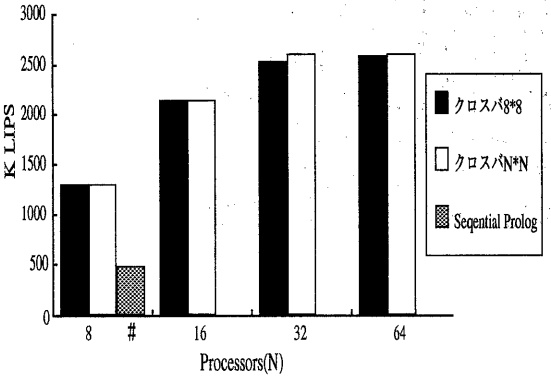


図８　プロセッサ台数と実行速度

図８によれば、2.5倍ほどのOR並列化の効果が認められる。また、プロセッサ台数の効果が見られるのは32台程度までであることが分かった。これは、プロセッサ台数が32台程度を超える場合は、既にアイドル状態のプロセッサがあるにもかかわらず、長い時間タスクが割り当てられないプロセッサが多数あることを意味している。

そこで、プロセッサ台数、環境フレームモジュールが、それぞれ8、32で、ネットワークが8×8の場合に8-QUEEN問題を実行したときの、各プロセッサの稼働率を図９、図１０に示す。図９では、並列度が増すに連れて稼働率も増加し理想的な処理を行っていることがわかる。図１０では、並列度が増すに連れて稼働率も安定に近づくが十分とは言い切れない。両者を比較すると、プロセッサ台数を増加して高速化を得ようとすると、プロセッサの稼働率が低下することが分かる。これは、図８の結果とも一致している。

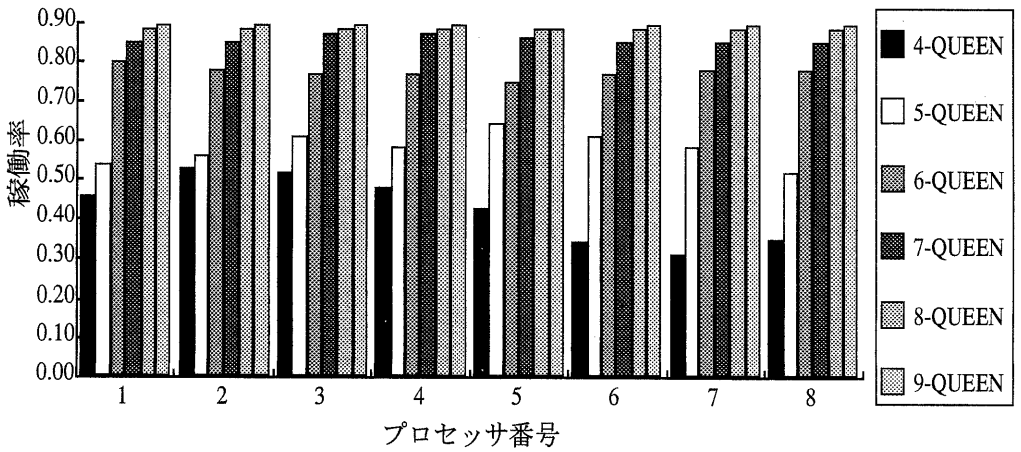


図９　プロセッサの稼働率（８台）

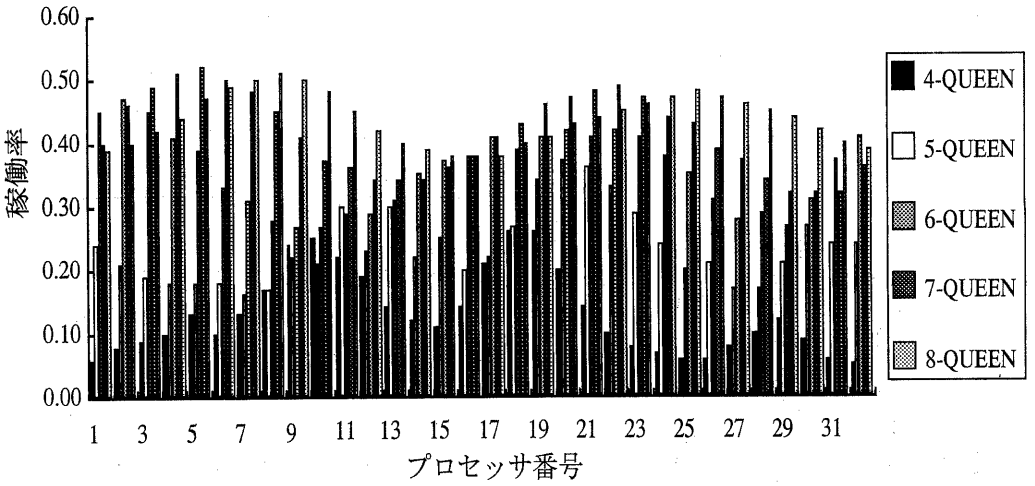


図１０　プロセッサの稼働率（３２台）

6. まとめ

本稿では、OR並列Prologを、パイプライン型アーキテクチャにマッピングする手法を提案し、その性能評価を行った。N-QUEEN問題でのシミュレーション結果によれば、メモリアクセス時のバス衝突によるオーバーヘッドは意外に少なく、むしろ処理の分散法に問題があることが分かった。逐次Prologのパイプライン処理[2]に対する、バス衝突によるオーバーヘッドの低下は、メモリ構造の変更や、ハッシュウインドウ法の導入による効果が大きいといえる。図７のネットワーク稼働率の結果から、今後、選択点スタックモジュールを２つのモジュールに分割して管理する手法を考えている。

また、本稿で提案した負荷分散法では、コンパイル時の静的な部分が主であった。図９での結果を見ると、本静的負荷分散法は有効であると思われる。しかし、静的な負荷分散のみでは限界があるため、動的なスケジューリング法の導入が必要である。一つの考えとしてはグローバルなパイプラインバッファを用いる方法が考えられる。これは、多量のタスクを生産し自分のパイプラインバッファに書き込めなくなったプロセッサが、グローバルなパイプラインバッファへの書き込みを行い、アイドルプロセッサがそのデータをもとに実行を開始する方法である。

さらに、今回のシミュレーションではコードブロックレベルのパイプライン処理を実現できなかったため、細粒度の並列性を取り出すことができなかった。今後は、コードブロックレベルのパイプライン処理を実現し、様々なプログラムでのシミュレーション評価を行う予定である。

参考文献

[1] Warren , D.H.D, "An Abstract Prolog Instruction Sets," Technical Report 309, AI Center, SRI International (1987).

[2] Joachim Beer, "Lecture Notes in Computer Science, " Springer-Verlag (1989).

[3] 市吉伸行, ” 論理型言語の並列処理方式”, 情報処理vol.32 No.4 (1991).

[4] Westphal,H., Robert,P., Chassin de Kergommeaux, J. and Syre , J.-C., "The PEPSys Model : Combining Backtracking, And-Or parallelism," In Proceedings of SLP (1987).